# KERNEL IMPLEMENTATION OF MONITORS FOR ERIKA

**by**

**SATHISH KUMAR R. YENNA**

B.Tech., Jawaharlal Nehru Technological University, India, 2001

---

A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

College of Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

2003

Approved By

Major Professor

Mitchell L. Neilsen

This document is prepared using LaTeX.

# ABSTRACT

ERIKA (Embedded Real tIme Kernel Architecture) is a research project on micro-kernel architectures for embedded systems. Users can develop embedded applications on ERIKA without worrying about the underlying architectural and implementation details. Monitors are program modules that provide a structured approach for process synchronization and can be implemented as efficiently as semaphores. ERIKA presently supports basic mutual exclusion primitives and semaphores for thread synchronization. It does not support monitors, which provide an efficient data abstraction mechanism. This thesis is about implementing monitors for ERIKA using the available kernel primitives. This implementation enables users to directly declare, define and use monitors in their applications.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGEMENTS

First of all, I would like to thank my major professor, Dr. Mitch Neilsen, for allowing me to do this thesis under his guidance. I am grateful to him for all the advice, encouragement and support he provided me with. I wish to thank him for believing in me and providing me the opportunity to work with him as a Graduate Research Assistant.

Next, I would like to thank Dr. Masaaki Mizuno, for being a continual source of inspiration and encouragement, and Dr. Gurdip Singh, for kindly consenting to be on my committee.

Also, I want to thank my parents, for always believing in me and for allowing me to take chances, and I thank my friends for their invaluable support and for their warm wishes.

Finally, I thank God for giving me the oppportunity to pursue a dream, and for always being there for me when I needed.

# DEDICATION

*To My Parents*

# Chapter 1

# Introduction to ERIKA

ERIKA (Embedded Real tIme Kernel Architecture) [1] is a research project on micro-kernel architectures for embedded systems. ERIKA is composed of a set of kernels and a set of tools:

- a scheduling simulator, that can be used to simulate the scheduling behavior of the system

- a set of optimized algorithms, which help the designer to find an optimal assignment of scheduling parameters that minimize the RAM usage while maintaining the schedulability of the system.

ERIKA Kernels consist of two layers: the Kernel Layer and the Hardware Abstraction Layer (HAL). The Kernel Layer contains a set of modules that implement task management and real-time scheduling policies. The Hardware Abstraction Layer contains the hardware dependent code that handles context switches and interrupt handling. Currently, there are HALs for the Seimens ST10/C167 architecture and for the ARM7TDMI architecture. It is easy to port ERIKA to a new processor family in a very short time by writing the HAL routines for the new architecture.

The interface is identical for all the implementations of the operating system, in order to allow portability and re-usability of application software.

## 1.1 Architecture

The following paragraphs describe the architecture of the ERIKA from a functional point of view.

A Kernel represents the core of a Real Time Operating System, which is a software layer that directly interacts with the hardware on which the application program runs. The main purpose of the Kernel is to give the application an interface capable to handle a set of entities that tries to abstract from the peculiarities of the underlying hardware.

The ERIKA kernel is able to

- Handle threads (a thread is a function which is executed concurrently with other entities in the system)

- Handle synchronization between threads (threads can share information by communicating with each other)

- Handle interrupts (an interrupt is a mechanism used to signal a running thread asynchronously)

## 1.2 Thread Handling

A multiprogramming environment provides entities that can execute concurrently. These entities are called threads, processes, and tasks. The term process is usually

used to identify an address space with one or more threads executing within that address space. The term thread can be thought as a single flow of control within a process. The term task is used in the real-time literature with different meanings. ERIKA does not support any kind of memory protection, so we can say that the ERIKA kernel can handle a set of threads. The ERIKA Operating System can be thought as a single process with many threads executing within the same address space.

The ERIKA kernel provides only the basic features for thread handling: it implements a real-time scheduling algorithm, a thread activation service and all of the underlying context switching functions.

Each thread owns a set of attributes (body function, priority, user stack, registers, etc.) that identify the state of the thread. Each thread is associated with a status word, which describes the state of the thread with respect to the scheduling algorithm. In ERIKA a thread can be in one of the following four states:

- *Stacked:* The thread, $t$ started its execution, and its frame is currently allocated on its stack; that is, $t$ is the running thread or $t$ has been preempted by another thread.

- *Ready:* The task has been activated and is ready to execute. All ready tasks are queued in a data structure called ready queue.

- *Idle:* The last instance of thread $t$ has terminated and $t$ is not queued on the ready queue. Moreover, $t$ does not have any memory allocated on its stack. A task in the idle state is usually is waiting for activation.

- *Blocked:* This state is used when a task is blocked on a synchronizing condition. Note that this state is meaningful only in a multi-stack HAL, where a thread can be blocked without interrupting others.

## 1.3 Thread Synchronization

The ERIKA kernel supports some synchronization mechanisms that can be used to guarantee that some critical code sections are executed in mutual exclusion, and other synchronization mechanisms that can be used for thread communication.

ERIKA system provides a type *MUTEX* and two mutex primitives, *mutex_lock()* and *mutex_unlock()*, that can be used to set the boundaries of critical regions. There are also blocking primitives implemented, such as classic semaphores and Cyclical Asynchronous Buffers (CABs). The ERIKA distribution does not support monitors, which is the main topic of this document.

## 1.4 Interrupt Handling

The ERIKA kernel can be configured to directly interact with the underlying hardware architecture. In particular, it can handle the interrupts that are raised by the I/O interfaces, timers, and so on in a clean way. In fact it provides an interface that allows linking a handler written by the user into an interrupt vector. That interface also provides proper handling of eventual interrupt controllers that may be present in a particular architecture.

The handler can be considered a special function that can call a subset of the primitives. These primitives can be used to interact with the operating system, for example, activating a thread that will terminate the interaction with the hardware interface. In this way, urgent pieces of code can be put directly into the handler. On the other hand, less urgent tasks can be processed in a separated thread without interfering the high priority tasks.

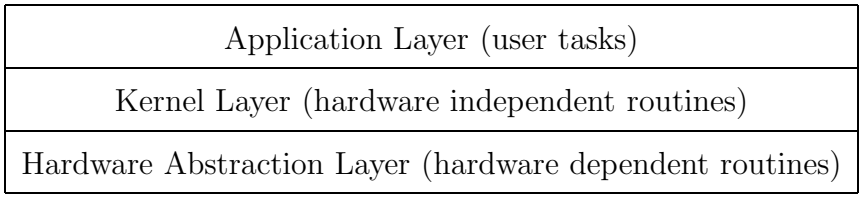
## 1.5 Layers

Two main layers compose the ERIKA kernel:

**Kernel Layer:** This layer is the software layer that exports all of the system primitives to the application. It is written using the C programming language in a way independent from the underlying architecture. This layer uses the services offered by the HAL for thread and hardware interface handling.

**Hardware Abstraction Layer (HAL):** HAL is the software layer used by the Kernel Layer to abstract from a particular architecture. This is the only nonportable part of the system, and it isolates all the peculiarities of the underlying architecture from the implementation of the behavior of the primitives. All HAL primitives are related to low-level aspects like context switching and interrupt handling. All other levels use the primitives provided by the HAL. Porting the ERIKA kernel to another architecture requires only modification of the HAL.

Figure 1.1: Layers of ERIKA.




### 1.5.1 The Kernel Layer

The functions provided by the Kernel Layer can be grouped in:

***Queue Handling:*** The kernel uses some queue data structures to keep track of the state of all threads in an application. The functions for queue management cannot be called directly by the application, but only from inside the

kernel. The queues are usually ordered.

***Scheduling:*** The kernel uses some extra functions to schedule the threads in the system. These functions use the HAL interface and the queue handling functions. These functions are not directly accessible to the user.

***User Interface:*** This is the part of the kernel that exports the constants, types, data and primitives the user can use. The user interface, in particular, covers all system services that are handled by the kernel.

The kernel layer internally uses some extra information about the tasks. In particular, it defines for each thread the status, the priorities, and the pending activations. Moreover it defines the mechanisms for mutual exclusion and the available resources.

### 1.5.2 The Hardware Abstraction Layer

The main objective of the Hardware Abstraction Layer is to export to the Kernel Layer the abstraction of thread. In general, each thread is characterized by:

- *A body*, which is a C function that implements the control flow of the thread. This function is called by the HAL with interrupts enabled each time a particular thread becomes the running thread.

- *A context*, which is the CPU status of the thread at a particular instant in time. The real composition of the context varies from architecture to architecture. The context can be thought as a copy of the CPU registers that is made each time the running thread is preempted or interrupted for some reason. In any case the application can not rely on the composition of a context: in general a particular HAL can save only a subset of the CPU

registers; that is, the minimum set of registers that can reproduce the status of a thread without any problem.

In practice, the HAL offers support for thread execution. Using the HAL interface, threads can be executed or stopped, and their context saved. The Kernel Layer internally only refers to a thread using some identifier called a Thread IDentifier (TID). This allows the implementation of various scheduling algorithms in a way independent of the particular hardware architecture.

The thread execution model is one shot: in that model, the body function simply represents an instance of a thread. Every time a thread is activated, the body function is called; when the body function finishes, the thread instance is terminated. Note that when an instance terminates, the end of the body function automatically clean up the stack frame of the thread. Typically the end of a body function jumps to a HAL/Kernel Layer function that is responsible for choosing the next thread to execute.

The memory model supported by the HAL is a shared memory model, where the whole address space is shared among all threads in the system. This assumption that reflects in fact the typical hardware architecture of most microcontrollers that typically do not support memory protection (usually they do not have a MMU).

The interface exported by the HAL, and the one-shot thread model allow in general different threads to share the same stack. Anyway, stack sharing can only be exploited if a proper scheduling algorithm is used. In general, that scheduling algorithm must ensure that once a task is started and its frame is allocated on the stack, none of the tasks with lower priority, and with which the task shares the stack, can be executed. An example of this algorithm is, for example, the Stack Resource Policy that allows resource and task sharing, and prevents deadlocks.

Depending on the scheduling algorithm used by the kernel, a proper HAL should be used. For example, if a Kernel Layer that allows tasks to block (e.g. providing blocking semaphores for synchronization) the HAL not only has to allow stack sharing but also allows different tasks to execute concurrently on different stacks in an interleaved way. Supporting a model that allows the use of more than one stack can lead to a degradation in performance for those applications that do not need different stacks, and where all the threads share the same stack.

For that reason, the ERIKA Project provides more than one HAL for each architecture, where each HAL is optimized for a particular usage; i.e., for mono and multi stack applications:

**Mono-Stack HALs [2]** All the threads and the interrupts share the same stack. This kind of HAL can only be used if the scheduling algorithm used by the Kernel Layer guarantees that a preempted thread cannot execute again before all the preempted threads have finished their instances.

**Multi-Stack HALs [3]** Every thread can have its private stack, or it can share it with other threads. This allows the Kernel Layer to use scheduling algorithms that could block threads in synchronization primitives (as for example, a fixed priority scheduler with semaphores) or for arbitrary reasons (as, for example, a classical round robin scheduler). Note that this model can also adapt to architectures that support memory protection, since every thread stack must stay in different address spaces.

Please note that the mono-stack HAL behavior can be viewed as a particular case of the multi-stack HAL behavior, where all tasks shares the same stack. The main differences between the two versions are the execution time and memory required to perform the context switches (the mono-stack HAL should be in general

preferred if possible).

Finally the HAL, besides introducing the thread abstraction, offer a general abstraction of the peripherals provided by a particular architecture. This generally includes a set of functions that allow the application to interact with a particular interface, together with the definition of a user handler to be associated with each interrupt.

## 1.6 System Modules

One of the objectives when designing an embedded application for systems that need a reduced memory footprint is to precisely tune which parts of the OS[1] should be included in the final system, and which not. In this way, both execution performance and memory footprint can be optimized.

The design of ERIKA helps the process of optimizing the memory foot print, because each part of the system can be thought as a module that can be included or not in the final image file. The entire Kernel is in fact compiled and linked together with the application, reducing every possible overhead.

In particular, the user can choose the following options at compile time:

- The hardware architecture on which the application will execute.

- The HAL to be used (mono-stack or multi-stack).

- The scheduling algorithm used by the application.

- Other external modules (for example, semaphores, mailboxes, and so on).

Finally, the application will define the initialization of the HAL and kernel data

[1]Operating System

structures that are application dependent (for example, the number of threads and their priority).

## 1.7 Code Portability

The division of the RTOS in two layers eases the porting of the Kernel Layer on top of different architectures. The use of a common interface for handling operating system calls also ease the migration from one scheduling algorithm to another one. Moreover, the interaction between the application and the operating system is done using the syntax of a high level programming language, which allows the specification of the constraints in a programming environment independent of the architecture.

# Chapter 2

# Kernel API

Kernel Layer provides an interface for the application. An application can directly call the kernel primitives to get services provided by the kernel. The following kernel primitives can be used in an application.

Table 2.1: Kernel primitives that can be used in an application [1].

| | |
|---|---|
| Thread Management | *thread_make_ready()* |
| | *thread_activate()* |
| | *sys_scheduler()* |
| | *thread_end_instance()* |
| Thread Management into Interrupt Drivers | *IRQ_make_ready()* |
| | *IRQ_end_instance()* |
| | *set_handler()* |
| Shared Resource Management | *mutex_lock()* |
| | *mutex_unlock()* |
| Utility Functions | *sys_gettime()* |
| | *sys_panic()* |
| | *sys_reboot()* |
| | *sys_idle()* |

## 2.1 Thread management

Each thread in the system owns a flag that contains information about its state. Recall that a thread can be in one of the following four states: stacked, ready, idle, or blocked.

In general the transition of a thread from a state to another one is triggered by the execution of a kernel primitive.

- ***Thread Activation:*** Some primitives can activate a thread. In this case, the activated thread jumps in the ready state waiting the availability of its execution. If the activated thread is already in the ready or stacked state, its current state is not changed, and the activation is stored in a pending activation counter.

- ***Thread Wakeup:*** Some primitives can wake up a thread that is already activated and that was not executing for some reason. For example, a thread could be waiting for a synchronization point.

- ***Preemption Check:*** Most of the kernel primitives include in their behavior a check that is used to guarantee that the highest priority always preempt lower priority threads.

The Figure 2.1 summarizes all the possible state changes and the kernel primitives that cause them.

In the following paragraphs, the kernel primitives are detailed.

***void thread_activate(TID thread)***

This primitive activates the thread whose index is passed as parameter. As a result the activated thread is put into the ready state if the running thread has higher

priority. If not, the activated thread becomes the running thread preempting the thread that called the primitive. If the activated thread is already active, its state is not changed and a pending activation is accounted for it.

***void thread_make_ready(TID thread, TYPENACT nact)***

This primitive tells the kernel that the thread passed as first argument will, when possible, execute for *nact* instances. This primitive never blocks the running thread, it simply increments the pending activation counter. An activated thread with priority higher than the running thread will execute after the next preemption check.

***void sys_scheduler(void)***

This primitive simply execute a preemption check to see if the highest priority thread is in the ready queue has higher priority than the running thread. In that case the higher priority thread in the ready queue preempts the running thread.

***void thread_end_instance(void)***

This primitive is called to terminate the current instance of a thread. This must be the last call in the thread, and it must be called at the same level as the '{' and '}' C symbols that delimit the thread body. If the thread body does not end with this call, results can be unpredictable.

Context change scenarios due to the activation of a thread that has higher priority with respect to the running thread using *thread_activate()* and *thread_make_ready()* are shown in the Figures 2.2 and 2.3.

## 2.2 Thread Management in Interrupt Handlers

An interrupt handler is a routine written by the user and executed to handle the arrival of an interrupt. Interrupt handlers are executed in special contexts that can be different from the context of the running thread that the handler interrupted. For that reason an interrupt handler can only call the primitives of this class. These primitives are architecture dependent though the names are same.

***void IRQ_make_ready(TID thread, TYPENACT nact)***

This function works as *thread_make_ready()* primitive.

***void IRQ_end_instance(void)***

This primitive must be called as the last instruction of an interrupt handler. The behavior of this function is the termination of the routine. Usually this function also does the preemption check.

## 2.3 Shared Resource Handling

The kernel layer provides a set of functions that can be used to enforce mutual exclusion between threads. These functions manages mutexes, that are basically binary semaphores statically defined and initialized.

The kernel layer provides two functions for managing mutexes, that are *mutex_lock()* and *mutex_unlock()*. Every critical section that use mutexes must start locking a mutex and must end unlocking it.

***void mutex_lock(MUTEX mutex_id)***

This primitive locks a mutex not already locked by another thread. Nested locks

of the same mutex in the same thread are prohibited.

***void mutex_unlock(MUTEX mutex_id)***

This primitive frees a mutex that was previously locked by the same thread. The unlocking of the mutex usually provokes a preemption check.

## 2.4 Utility Functions

This section shortly describes a few functions that can be used in the application.

***void sys_panic(void)***

This function is invoked in case of an abnormal state of the application. Typically this function should reset the system and/or signal a system fault.

***void sys_reboot(void)***

This primitive resets the system.

***void sys_idle(void)***

This function is the function that should be used into the idle loop in the *dummy()* function. This function does nothing.

***void sys_gettime(TIME *t)***

This function returns the timer value that is currently used as system clock. This function is only available if the user defines the symbol __TIME_SUPPORT__.

## 2.5 The dummy() thread

The startup point of an application is a function called *dummy()* whith the prototype *void dummy(void)*.

The dummy thread can be thought as the thread that is responsible for system initialization and the idle loop. The dummy thread is always the lowest priority thread in the system, it does not have a TID, and it never ends. It is always active and for that reason it can not be activated. It is used for:

- **System initialization:** The *dummy()* thread should initialize all peripherals.

- **Thread activation:** It should also manage the initial thread activations through the use of the *thread_make_ready()* primitive.

- **Preemption:** The *dummy()* function usually calls the *sys_scheduler()* or *thread_activate()* primitive to execute a preemption check. Otherwise, no other thread is activated and the system will be in the infinite idle loop.

- **Idle loop:** The *dummy()* function should execute an infinte idle loop. During the idle loop any kind of background code can be executed; if no code to be executed, the *sys_idle()* primitive should be called.

Figure 2.1: Thread Transition Diagram [1].

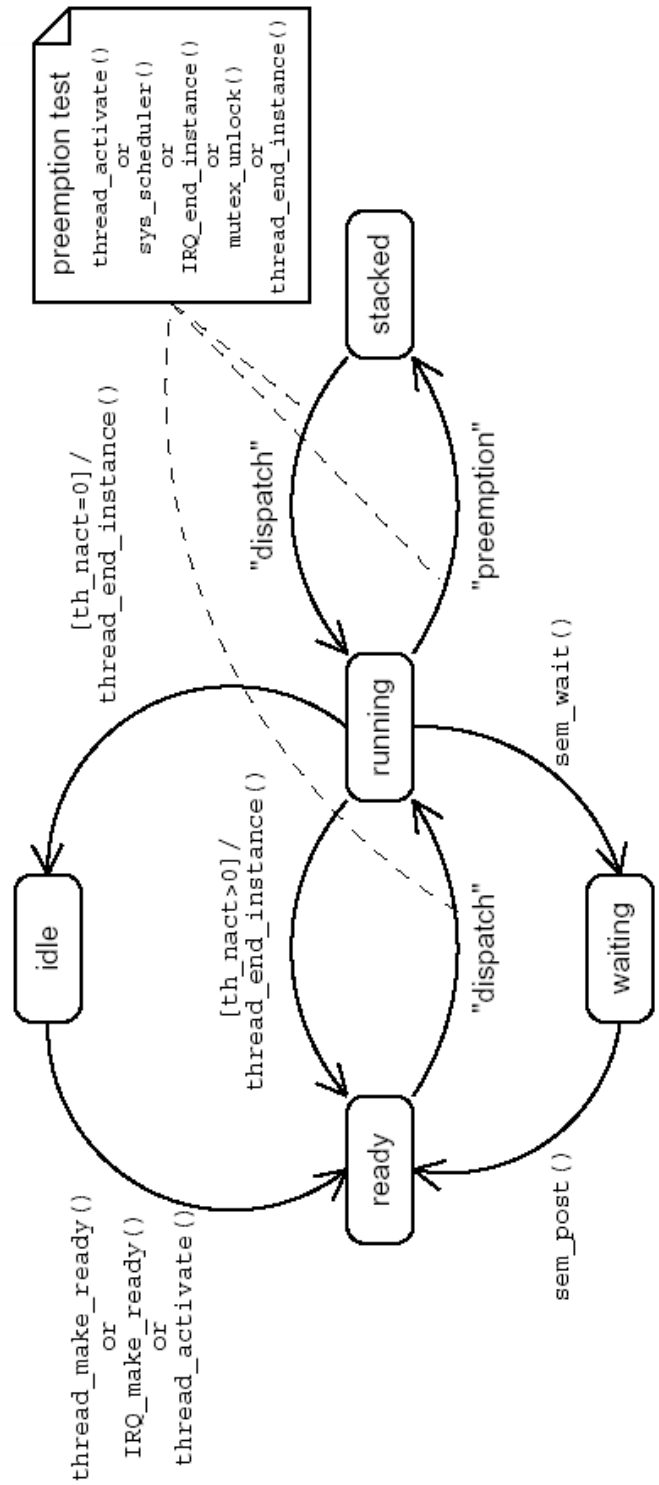


Figure 2.2: Context changes due to the activation of thread B that has higher priority with respect to the running thread A [1].

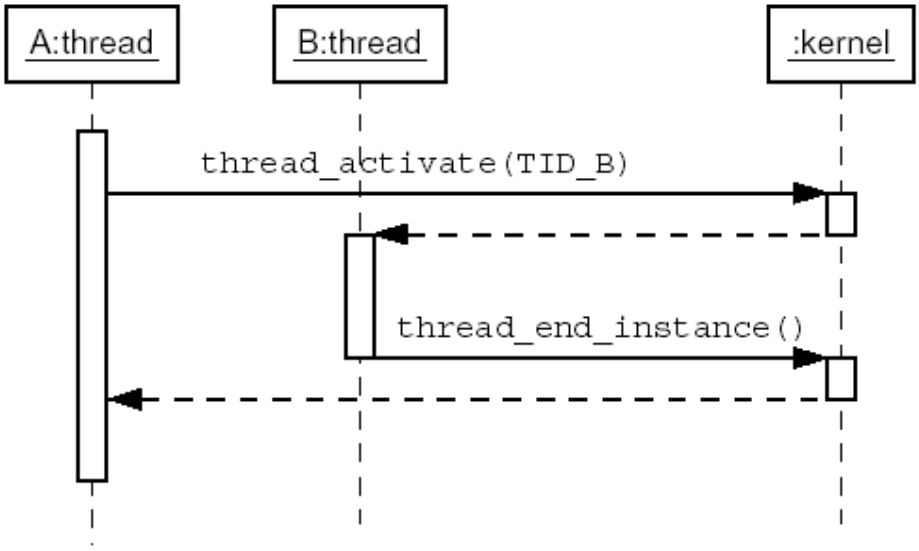
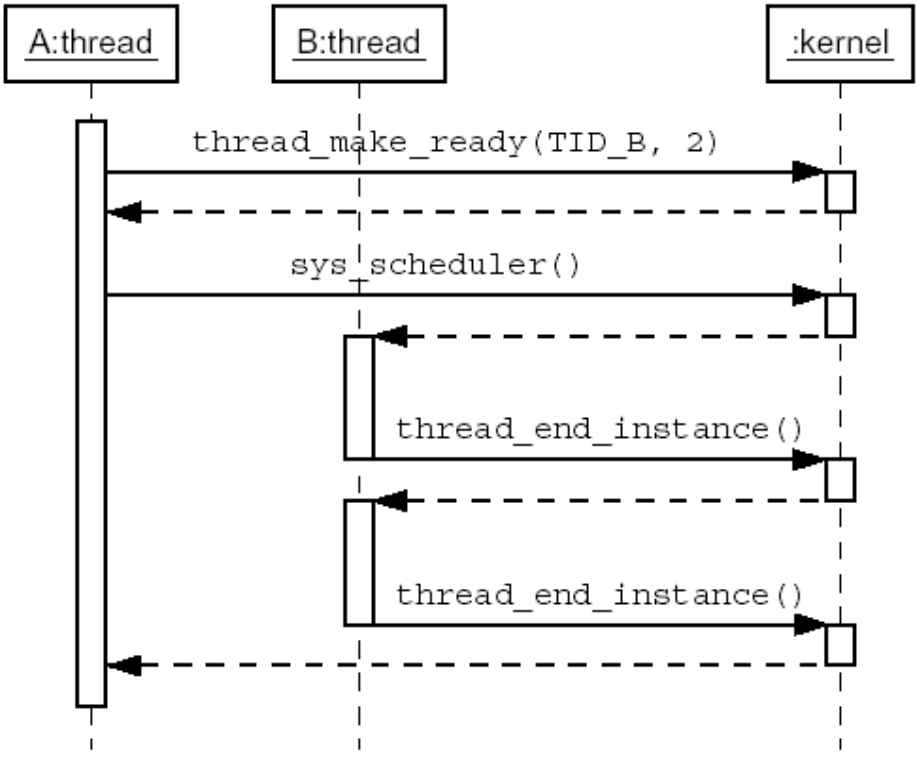


Figure 2.3: Context switches due to the activation of thread B by a call of *thread_make_ready()* by a lower priority thread A [1].



# Chapter 3

# Kernels

ERIKA currently provides two kernels: FP (Fixed Priority scheduling with Preemption Thresholds and SRP[1] protocols) and SRPT[2] (Earliest deadline scheduling with Preemption Thresholds and SRP protocols).

## 3.1 FP Kernel

The scheduling algorithm used by the FP Kernel is the Fixed Priority with Preemption Threshold Scheduling Algorithm. Each thread in the system has a priority called ready priority that is used to enqueue threads in the ready queue. When a task becomes the running task, its priority is raised to another priority level, called dispatch priority (that usually is greater than the ready priority). Lower priority threads do not execute until there are no higher priority threads ready for execution, so at any instant of time the thread with the highest priority is the running thread.

Threads can share resources, and mutexes are used to enforce mutual exclusion

[1]Stack Resource Policy

[2]Stack Resource Policy with Preemption Threshold Scheduling

between them. Each mutex has assigned a ceiling that is the maximum ready priority of the threads that use that mutex. When a mutex is locked, the priority of the task is raised to the mutex ceiling.

The protocol just showed avoids deadlocks, chained blocking and priority inversion. Moreover, all the threads can share the same stack, and dispatch priorities can be used to reduce the preemption between threads in a controlled way, further reducing the overall stack space needed.

### 3.1.1 An Example

As an example, in this section we depict a typical situation that explains the internal mechanisms of the kernel. Suppose to have the thread set depicted in Table 3.1 with the corresponding initialization values. The thread set is composed of six threads, each with different priority; there is a non-preemption group composed by thread 4 and 5 (they have the same dispatch priority), and there is a shared resource used by threads 1 and 3 through a mutex. The system ceiling[3] has a starting value of 0x01 (because the thread 1 starts on the stack).

Table 3.1: The example thread set [1].

| Thread | Ready | Dispatch | Initial Values | | | After the events | | |
|---|---|---|---|---|---|---|---|---|
| Number | Priority | Priority | Status | Pend. Act. | next | Status | Pend. Act. | next |
| 0 | 0x01 | 0x01 | READY | 0 | NIL | STACKED | 1 | NIL |
| 1 | 0x02 | 0x02 | READY | 0 | NIL | STACKED | 1 | 0 |
| 2 | 0x04 | 0x04 | READY | 0 | NIL | READY | 0 | NIL |
| 3 | 0x08 | 0x08 | READY | 0 | NIL | READY | 1 | NIL |
| 4 | 0x10 | 0x20 | READY | 0 | NIL | STACKED | 1 | 1 |
| 5 | 0x20 | 0x20 | READY | 0 | NIL | READY | 1 | 3 |

[3]As there are only 6 threads in this example, only the first 8 bits are showed; the MSB is considered equal to 0x00.

Then suppose that these events happen in the system:

1. System start: Thread 0 is activated, so it starts in the STACKED state;
2. Thread 1 arrives and preempts Thread 0 since it has a higher priority;
3. Thread 1 locks the mutex; therefore, the system ceiling is updated to 0x0B;
4. Thread 3 arrives. Even though its priority is greater than the priority of Thread 1, it is not greater than the current system ceiling: hence it goes in the ready queue;
5. Thread 4 arrives and become the running thread;
6. Thread 5 arrives and since it belongs to the same nonpreemption group of Thread 4 and Thread 4 is still active, then it goes into the ready queue.

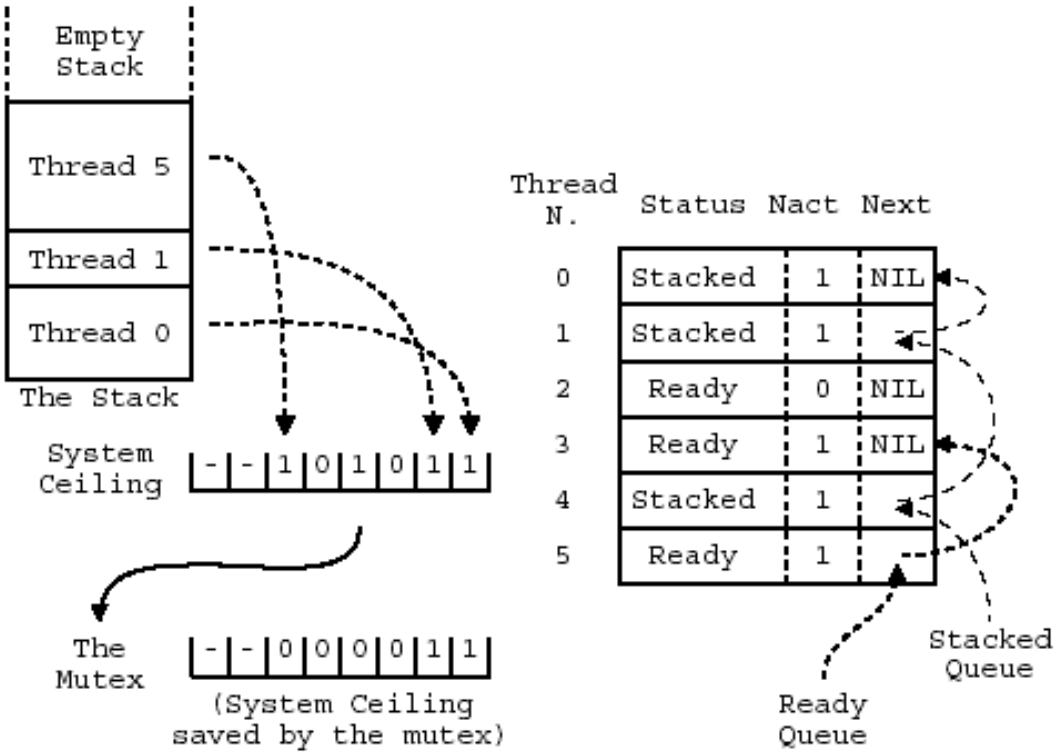
Finally, suppose that all the threads share the same system stack. After these events, the configuration of the kernel data structures is shown in the Figure 3.1.

## 3.2 SRPT Kernel

The scheduling algorithm used by the SRPT Kernel is the Stack Resource Policy with Preemption Threshold Scheduling Algorithm. This algorithm is very similar to the one used in the FP Kernel, except that it uses Earliest Deadline First (EDF) scheduling instead of a Fixed Priority assignment.

Each thread in the system has a deadline that is used to enqueue threads in the ready queue. Moreover, each thread has a preemption level that is chosen proportional to the inverse of its period, and a threshold. The system maintains a system ceiling, which stores the thresholds of the stacked threads. When a task

Figure 3.1: Kernel Data structures snapshot after the events listed [1].




becomes the running task, the system ceiling is raised to at least the threshold of the running task.

Threads can share resources, and mutexes are used to enforce mutual exclusion between them. Each mutex has assigned a ceiling, which is the maximum preemption level of the threads that use that mutex. When a mutex is locked, the system ceiling is raised at least to the mutex ceiling. At any instant of time, a thread can become the running thread only if it has the earliest deadline, and its preemption level is greater than the system ceiling.

The protocol just showed avoids deadlocks, chained blocking and priority inversion. Moreover, all the threads can share the same stack, and dispatch priorities

can be used to reduce the preemption between threads in a controlled way, further reducing the overall stack space needed.

# Chapter 4

# HAL Internals

One of the main design issues in the design of a Kernel Layer is the code portability, in the sense of the ability of reusing all the Kernel Layer source code in more than one architecture. For that reason, all the code of the Kernel Layer is standard C code, whereas all the architecture dependent issues are hidden under the HAL Layer interface.

Note that applications never call HAL functions directly. They can only call the primitives provided by the Kernel Layer independent of their implementation in the HAL layer.

There are two reasons why a user can not call directly the implementation provided by a kernel:

- Every architecture has its own methods for calling a primitive, and a Kernel implementation should only implement the behavior of a particular primitive, without relying on nothing else than the standard C HAL interface.

- Some CPU architectures allow an efficient implementation of some primitives. For example, the *mutex_lock()* primitive can be implemented using ATOMIC instruction under ST10, giving an inline implementation that is as

fast as a simple function call.

## 4.1 HAL Interface

HAL Layer is a software layer that is used to abstract a particular implementation from the underlying hardware. HAL interface offers the following services:

- **Thread Abstraction:** A thread is identified by the HAL by a C function, and by the information that is stored in ROM and RAM, such as the stack space it occupies when it is stacked, the copy of CPU registers if needed and other implementation dependent information.
- **Context Handling:** The HAL implements a limited set of functions that can be used to implement context switching between tasks.
- **Interrupt Handling:** The HAL implements the entire interface needed to properly handle interrupts.
- **Utility Functions:** Finally, some functions for time, idle time, and reboot handling are provided.

All these services are not used directly by the application, but they are called through the services exported by the Kernel Layer.

The HAL interface is implemented by the HAL Layer. In general, HAL layer can be developed following two different paradigms, which in the following paragraphs we highlighted as monostack HAL and multistack HAL.

**Mono-Stack HAL** Following this model, all the threads share the same stack. This implicitly assumes that a preempted thread can not execute again until all the threads and interrupts handlers that preempted it have finished. Note that it is not possible to implement any kind of blocking primitive with this paradigm.

Table 4.1: The Basic HAL Interface [1].

| | |
|---|---|
| Context Handling | *hal_endcycle_ready()* |
| | *hal_endcycle_stacked()* |
| | *hal_ready2stacked()* |
| | *hal_IRQ_ready()* |
| | *hal_IRQ_stacked()* |
| | *hal_stkchange()* |
| Primitive Definition | *hal_begin_primitive()* |
| | *hal_end_primitive()* |
| | *hal_IRQ_begin_primitive()* |
| | *hal_IRQ_end_primitive()* |
| Interrupt Handling | *hal_enableIRQ()* |
| | *hal_disableIRQ()* |
| | *hal_IRQ_enableIRQ()* |
| | *hal_IRQ_disableIRQ()* |
| Utility Functions | *hal_gettime()* |
| | *hal_reboot()* |
| | *hal_panic()* |
| | *hal_idle()* |

**Multi-Stack HAL** This model HAL handles more than one stack. In particular, stack sharing is limited by assigning a set of thread to a particular stack. As a limit all the threads use a private stack, and no stack sharing is exploited at all. Note that the assignment of threads to stacks is usually done statically.

All the HAL implementations should guarantee a thread that has been interrupted will start again form the same point where it was interrupted. As for the Kernel Layer, the application should define and initialize the HAL data structures. These data structures heavily depend on the architecture. The initialization

of these data structures can be statically done in the variable initializers or can be done in the *dummy()* function.

## 4.2 Context Handling

***void hal_endcycle_ready(TID t)***

This function is called as the last function in primitives such as *thread_end_instance()*. It destroys the context allocated by the thread that is just ended, and creates the context of the thread $t$ passed as parameter. Then the thread $t$ is executed. In case of a multi-stack HAL this function can change the current stack.

***void hal_endcycle_stacked(TID t)***

This function destroys the context allocated by the thread that is just ended and then executes the thread $t$ whose context is supposed to be on top of a stack(that in a multi-stack HAL may not be the current stack).

***void hal_ready2stacked(TID t)***

This function is used to dispatch a ready task. This function suspends the running thread (saving its context on its stack), then it creates the thread $t$ on its stack, and finally it executes the body of the thread $t$. In case of a multi-stack HAL this function can change the current stack.

***void hal_IRQ_ready(TID t)***

It has the same meaning of *hal_endcycle_ready()*, but it is called only at the end of the last nested interrupt handler. It must be the last function call of the interrupt handler.

***void hal_IRQ_stacked(TID t)***

It has the same meaning of the *hal_IRQ_ready()* except that it does not create the context for the thread $t$ but it assumes that it is present on top of the stack.

***void hal_stkchange(TID t)***

This function is implemented only by the multistack HALs. It changes the running thread to another thread $t$ that is already on the top of another stack. This function always change the current context. This function is used by the kernel to implement the synchronization points for the blocking primitives.

## 4.3 Kernel Primitive Definition

***void hal_begin_primitive(void)***

This function must be called as the first instruction of every primitive, to prepare the environment for the kernel.

***void hal_end_primitive(void)***

This function must be called as the last instruction of every primitive, to restore the environment of the calling thread. Note that it is implementation defined if this function returns to the kernel primitive or to the calling thread.

***void hal_IRQ_begin_primitive(void)***

This function must be called as the first instruction of every primitive used in an interrupt handler, to prepare the environment for the kernel.

***void hal_IRQ_end_primitive(void)***

This function must be called as the last instruction of every primitive used in an interrupt handler, to restore the environment of the calling thread. Note that it is implementation defined if this function returns to the kernel primitive or to the calling thread.

## 4.4 Interrupt Handling

***void hal_enableIRQ(void)***

It enables all the interrupt sources. It can only be called in a thread.

***void hal_disableIRQ(void)***

It disables all the interrupt sources. It can only be called in a thread.

***void hal_IRQenableIRQ(void)***

It enables all the interrupt sources. It can only be called in an interrupt handler.

***void hal_IRQdisableIRQ(void)***

It disables all the interrupt sources. It can only be called in an interrupt handler.

## 4.5 Utility Functions

***void hal_gettime(TIME *t)***

It returns the value of the system timer. Note that this function is only available if the user defines the __TIME_SUPPORT__ symbol in the ERIKA makefile.

***void hal_panic(void)***

It is called by *sys_panic()* to put the system to a meaningful state. It usually resets the system or it hangs the system signaling an abnormal termination to the external world.

***void hal_reboot(void)***

This function is called by *sys_reboot()* to reset the system.

***void hal_idle(void)***

This function is called by *sys_idle()* to put the CPU in an idle low power state. CPU does nothing except waiting for an interrupt.

Figure 4.1: Interaction between the kernel layer primitives and the HAL functions for context handling [1].

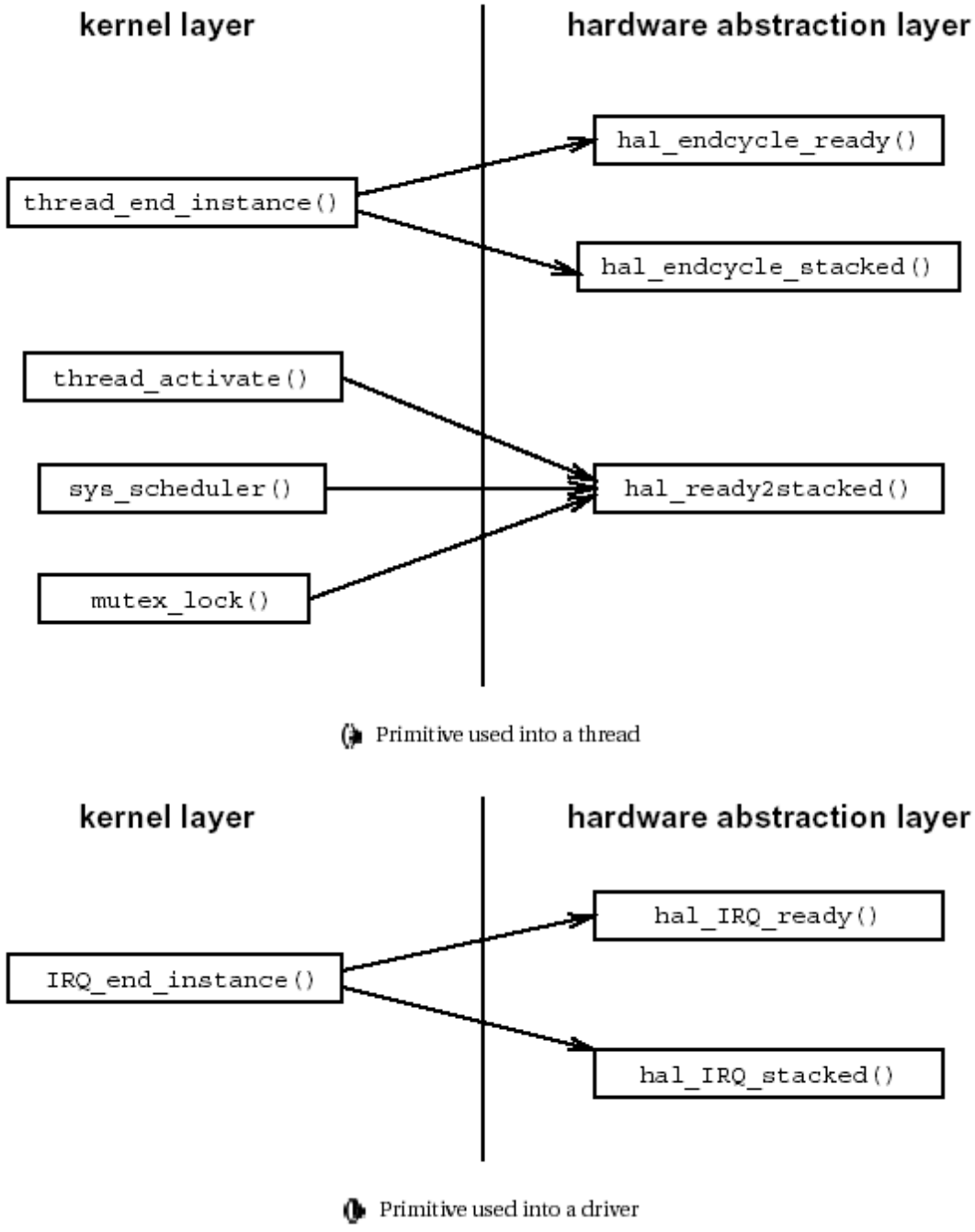


Figure 4.2: Context change due to the activation of a thread B at higher priority with respect to the running thread A [1].

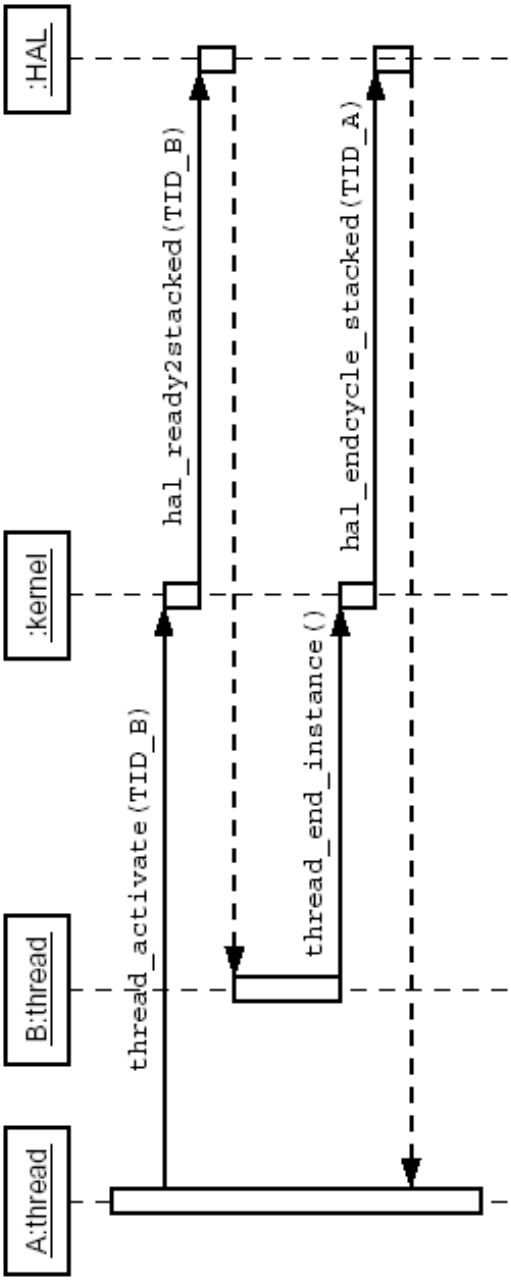
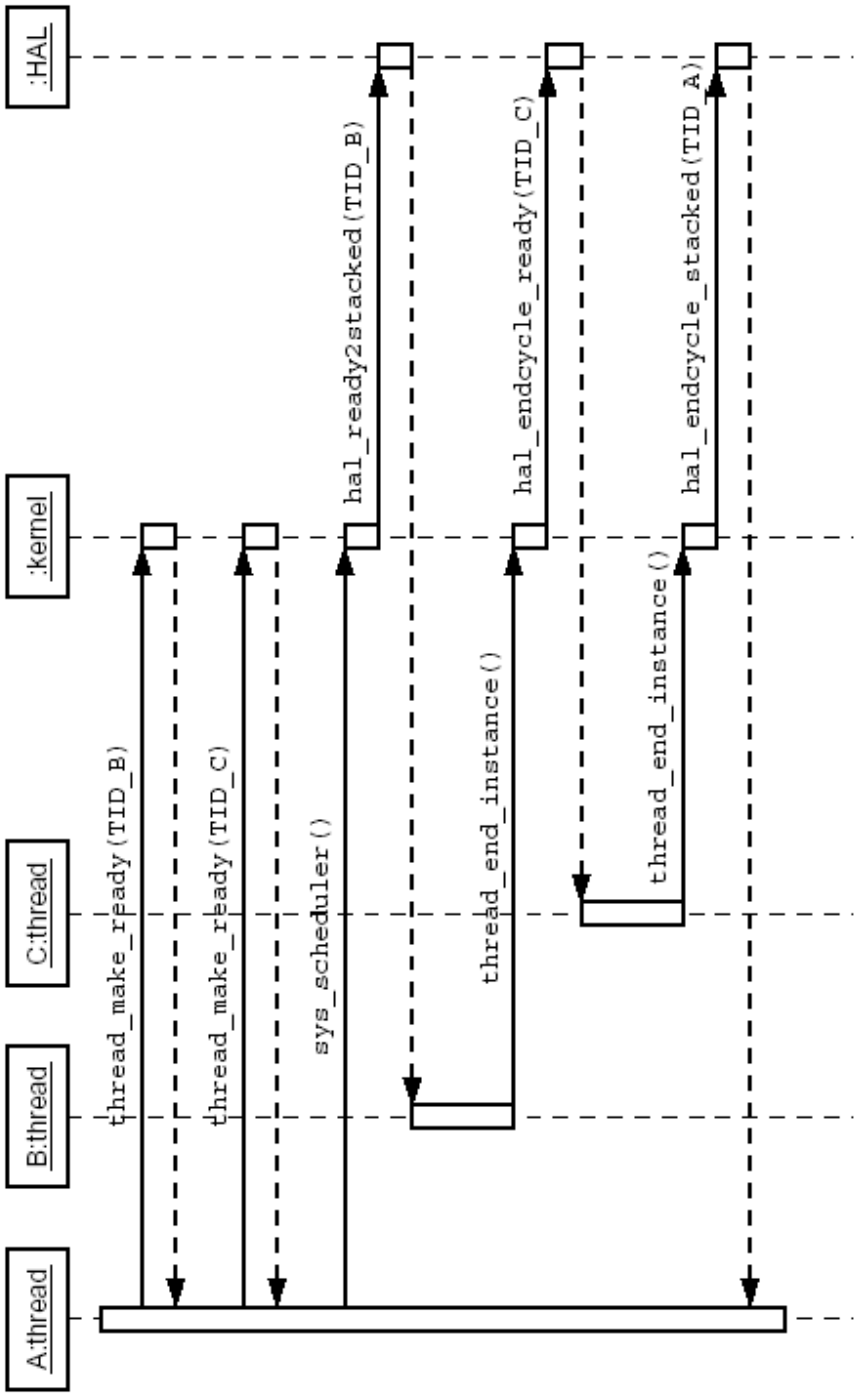


Figure 4.3: Context change caused by the activation of two higher priority threads with respect to the running thread [1].



# Chapter 5

# ST10 HAL Internals

This chapter describes some implementation details that can be useful to understand the architectural choices made to implement the ST10/C167 HALs. The ERIKA project provides three kinds of HALs for the ST10 platform. Each HAL has different requirements in terms of the underlying memory models and in terms of memory usage. Moreover, each HAL provide different performance in terms of execution time of the primitives.

**Mono-Stack HAL** This is the simplest and fastest HAL available for ST10. It supports only functions built for the system stack, and it is so simple that most of the HAL primitives can be expanded inline.

**Multi-Stack HAL** This is another HAL that is similar to the mono-stack HAL except that it supports multiple stacks allowing the usage of blocking primitives. We will look into details of this HAL since it is the simplest, efficient and sufficient for most of the real-time applications.

**Segmented Multi-Stack HAL** This HAL is the slowest one because it uses the Large Memory model and the user stack for function calls. It is useful only when the application source code is big and there is a need of using all the available

CPU memory.

Table 5.1: Characteristics of the ST10 HALs [1].

| Feature | Mono Stack | Multi Stack | Segmented Multi Stack |
|---|---|---|---|
| Speed | Fastest | Fast | Slow |
| Single Stack | Yes | No | No |
| Blocking Primitives Allowed | No | Yes | Yes |
| Memory Model | Tiny | Tiny | Large |
| Functions store return values into user stack | No | No | Yes |
| Typical ROM Footprint | XXX bytes | YYY bytes | ZZZ bytes |

## 5.1 Peculiarities of the ST10 Implementation

This section contains some notes about the implementation details that you can find in the source code.

***Compiler and configuration files*** The tool-chain used to compile the project is the TASKING tool-chain for ST10. The make file is done in a way that it will build all the source files of the Kernel into the out directory under the current application directory.

***System stack and user stack*** The ST10 architecture supports two stacks, a system stack and a user stack. The system stack is stored in the internal chip memory, and it is for this reason, fastest stack on machine. Unfortunately, its size is limited, so often the applications simulates a stack using the R0 register.

The mono and multi-stack HALs for reasons of speed do not support the use of the user stack for storing the return address of the functions. In other words, functions are called using the couple of assembler instructions CALL/RET.

The multi-stack kernel need to address more than one system stack. For that

reason the various system stacks are re-mapped in different places of the real system stack. Note that in this way there is no protection for system stack overflows. The different stacks are implemented saving at each stack switch the top of the user stack and of the system stack in a private HAL variable.

***CP register and register banks*** To speed up the kernel, a register bank is used accessible through the CP register for each preemption level. Every time there is a context change, the CP value is saved on the stack and is then restored.

Function calls, primitives and *__NOPT__* primtives can be thought as simple functions that disable the interrupts as their first operation. Interrupt disabling is a typical way of ensuring that the kernel and HAL data structures are accessed in mutual exclusion. A particular attention must be reserved to the end of a thread. In practice, a thread ends when it reaches the trailing '}' of its function definition. The RET instruction inserted by the compiler at the end of the function jumps to the internal function called *st10_thread_endinstance()*.This function cleans the stack and then depending on the priority and number of instances of the thread, *st10_thread_endinstance()* executes the right instructions to change the running thread accordingly to the kernel choice.

This way of implementing the end of a thread can be selected with the option *__NOPT__* in the *ERIKAOPT* variable in the make file. However note that the instructions that are needed to change the context are often only a few. For this reason the HALs support a way of ending the instance of a thread using an optimized inline substitution.

The advantage of the first approach is that the code produced by the compiler can be used straight without modification, allowing an easy extension of the Kernel Layer. If the maximum efficiency is needed, an the correspondent optimized version is available, the second approach can be used reducing code size and enhancing

the performance of the system.

***Timers and thread activations*** The ST10 architecture has a lot of nice features, including, in particular a lot of timers and CAPCOMs. The ERIKA implementation uses a free running timer(timer T8) to have a coherent timing reference, and the related CAPCOMs to raise the periodic interrupts in a coherent way.

***Interrupt Handlers*** Interrupt handlers are defined using the *hal_set_handler()* macro, that is responsible to link an interrupt source to a user function that will be called with interrupts disabled. Note that *hal_set_handler()* only set the function that have to be called when an interrupt arrives. The peripherals that raise the interrupts must be configured directly by the application.

## 5.2 Multi-Stack HAL

Under the multi-stack HAL the system is composed by a set of threads. Each thread has: a stack, a context, and a body. The thread stacks and context may be shared.

### 5.2.1 Internal data structures defined by the user

***iram ADDR st10_thread_body[]*** This array stores the thread body pointer for each thread in the system, because when a thread activation occurs, the HAL needs to know which is the body function to execute. These body pointers are stored in the HAL because the HAL interface only exports the concept of TID, so the kernel never needs to know about body functions.

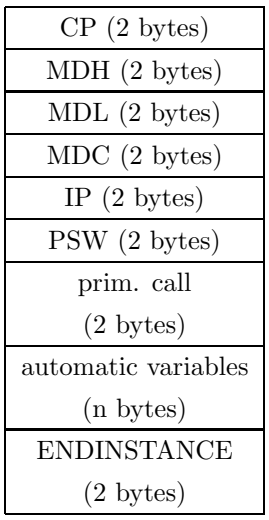
***iram ADDR st10_thread_cp[]*** This vector stores the register bank addresses that are used by the HAL for each thread.

***iram UINT16 st10_user_stack[]*** This vector is the user stack. The dimension of this vector should be enough to store all the user stack of the system.

### 5.2.2 Context Layout

Typical stack layout is as shown in the following figure.

Figure 5.1: Typical Stack layout of a preempted thread [1].




**ENDINSTANCE** This is the return address which the CPU jumps to when a thread instance ends. It is the address of the first instruction of the internal *st10_thread_endinstance()* function.

**Automatic variables** It is the space used by the compiler to allocate the automatic variables of a thread.

**Prim.call** This is the return address that is pushed on stack only if the thread was preempted after a call to some primitive (ex. *thread_activate()*) it is not present if the thread was preempted by an interrupt.

**PSW, IP** These values are pushed on the stack by a primitive or by the CPU interrupt response

**MDC, MDL, MDH** These are a copy of the ST10s multiply registers, and they need to be saved every time a thread is preempted.

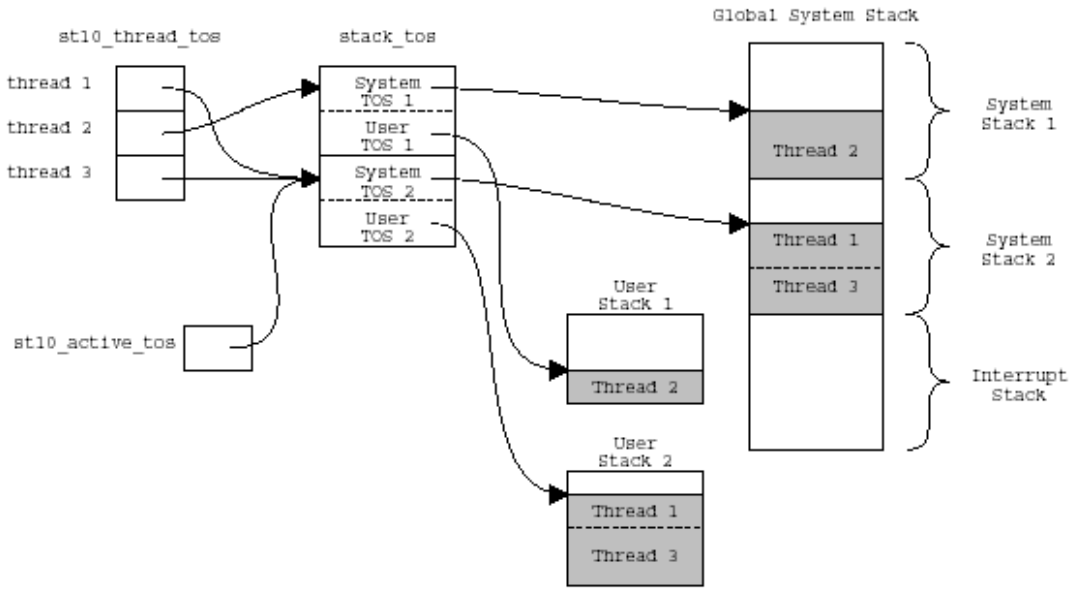
**CP** This is the St10's context pointer used by the thread. This is saved every-time a thread is preempted and it is restored every time preemption level to which the preempted thread belongs to.

### 5.2.3 Stack Implementation

To be compatible with the C compiler, the multi-stack HAL has to switch the user stack at each thread change. To allow maximum performance, the system stack is used for parameter passing, and the system stack of each thread is mapped in different location of the global system stack. No overflow check is done on the system stack.

A typical stack configuration can be depicted as shown in the Figure 5.2.

Figure 5.2: A typical stack configuration [1].



The important data structures are described in the following paragraphs.

***iram UINT16 st10_thread_tos[]*** This is an array that stores, for each thread, the addresses of the stack data structures. In the Figure 5.2 three threads are showed; thread 1 and thread 3 share the same stack. When a context change occurs, the HAL needs sometimes to know about the stacks of the threads. For this reason, the HAL interface is hacked adding a structure stack pointer.

***iram struct struct_tos stack_tos[]*** This array stores the information about each couple of stacks. Since each thread needs a system and a user stack, they are grouped in a structure called struct_tos, that contains the pointer to the top of the stack (tos) for a system stack and a user stack.

***iram WORD st10_active_tos*** This variable stores the stack used by the running thread. This variable must be used by the interrupt handler to save the tos before switching to the interrupt stack. In this way, the interrupt handler can rely only on the HAL implementation.

**Global system stack** The Figure 5.2 shows how the system stack is divided in slices. The situation shows how the three threads allocate its stack frames on their stacks, and shows also an empty interrupt stack (which starts and ends empty).

**User stacks** Finally, the Figure 5.2 shows the user stacks typically located in the external memory of the ST10.

### 5.2.4 Performance

The performance metrics of the Multi Stack ST10 HAL are presented in the Table 5.2. More observations are skipped for brevity.

Table 5.2: Performance metrics of the multi-stack ST10 kernel [1].

| | |
|---|---|
| *Kernel Data* | 250 bytes (RAM + RAM Initialization + ROM) |
| *Not-inlined code* | 794 bytes |
| *Test Application* | 132 bytes |
| *Timer Handler* | 176 bytes |
| *Total image of the sample application* | 1352 bytes (system stack, register banks and initialization not included) |
| *ROM Footprint* | 970 bytes + 10 bytes for each thread + 2 bytes for each SRP Mutex |
| *RAM Footprint* | 8 bytes global variables + 4 bytes for each thread + 32 for each priority level + 2 for each SRP Mutex + 4 for each stack + System Stack + User Stack |
| *System Stack Usage* | 16 bytes for each priority level + iterrupt frames |
| *Mutexes* | lock: 1.4 microseconds,unlock from 3.6 to 10.8 microseconds |
| *Activate* | from 2.8 to 9.6 microseconds |
| *End instance* | from 8.8 to 12.4 microseconds |
| *Interrupt Latency* | 6 microseconds |

# Chapter 6

# Monitors

Monitors [5] are program modules that provide a structured approach for process synchronization and can be implemented as efficiently as semaphores. First and foremost, monitors are a data abstraction mechanism: they encapsulate the representations of abstract resources and provide a set of operations that are the only means by which the representation is manipulated. In particular, a monitor contains variables that store the resource's state and procedures that implement operations on the resource. A process can access the variables in a monitor only by calling one of the monitor procedures. Mutual exclusion is provided by ensuring that execution of procedures in the same monitor is not overlapped. Condition synchronization in monitors is provided by a low-level mechanism called condition variables [4].

When monitors are used for synchronization, a concurrent program contains two kinds of modules: active processes and passive monitors. Assuming all the shared variables are within monitors, two processes can interact only by calling procedures in the same monitor. The resulting modularization has two benefits. First, a process that calls a monitor procedure can ignore how the procedure is im-

plemented; all that matters are the visible effects of calling the procedure. Second, the programmer of a monitor can ignore how or where the monitors procedures are used. Once a monitor is implemented correctly, it remains correct, independent of the number of processes that use it. Also, the programmer of a monitor is free to change the way in which the monitor is implemented, so long as the visible procedures and their effects are not changed. Together, these benefits make it possible to design each process and monitor relatively independently. This in turn makes a concurrent program easier to develop and understand [4].

## 6.1 Programming Using Monitors

A monitor places a strict boundary around both the variables and the procedures that implement a shared resource. A monitor declaration has the form:

*Monitor MonitorName*

*Declarations of shared variables; initialization code*

*Procedure operation1 (formal parameters)*

*Body of operation1*

*End*

. . .

. . .

*Procedure operationN (formal parameters)*

*Body of operationN*

*End*

*End*

The permanent variables describe the state of the resource. The procedures

implement the operations on the resource.

A monitor has three properties, which make it an abstract data type. First, only the procedure names are visible outside the monitor; they provide the only gates through the monitor boundary. To alter the state a process must call one of the available procedures. The second property is that the procedures within a monitor may access only the permanent variables and the local variables. They may not access the variables declared outside the monitor. Third, permanent variables are initialized before any procedure body is executed.

Processes executing in monitors may require mutual exclusion—to avoid interference; and may require condition synchronization—to delay until the monitor state is conducive to continued execution. We will now look at how processes synchronize within monitors.

## 6.2 Synchronization in Monitors

Synchronization is easiest to understand and hence to program if mutual exclusion and condition synchronization are provided in different ways [4]. It is best if mutual exclusion is implicitly provided since this precludes interference. By contrast, condition synchronization must be explicitly programmed since different programmers require different synchronization conditions.

Based on these considerations, mutual exclusion in monitors is implicitly provided and condition synchronization is explicitly programmed using mechanisms called condition variables. Since execution within a monitor is mutually exclusive, processes cannot interfere with each other when accessing permanent monitor variables.

A condition variable is used to delay a process that cannot safely continue

executing until the monitor's state satisfies some boolean condition. It is also used to awaken a delayed process when the condition becomes true. A condition variable is declared in the form

*var c: cond*

The value of $c$ is a queue of delayed processes, but this value is not visible to the programmer. The boolean delay condition is implicitly associated with the condition variable by the programmer. Since the condition variables are used to synchronize processes within monitors, they may be declared and used only within monitors.

To delay on a condition variable $c$, a process executes *wait(c)*. Execution of *wait* causes the executing process to delay at the rear of $c$'s queue. So that some other process can eventually enter the monitor to awaken the delayed process, execution of *wait()* also causes the process relinquish exclusive access to the monitor.

Processes waiting on some boolean condition associated with some condition variable $c$, are awakened by means of signal statements. If $c$'s delay queue is not empty, execution of *signal(c)* awakens the process at the front of the delay queue and removes it from the queue. That process executes at some time in the future when it can acquire exclusive access to the monitor. If $c$'s delay queue is empty, execution of signal has no effect.

Independent of whether a delayed process is awakened, the process executing signal retains exclusive access of the monitor; thus it can continue executing. Programmer has to take care of inserting *wait()* and *signal()* operations at the appropriate locations to avoid permanent blocking of processes. A process waiting on some condition variable must be awakened by some other process when its boolean condition is satisfied.

There is one more operation (Broadcast Signal) available on condition variables.

It is used if more than one delayed process can proceed or if the signaling process doesn't know which delayed process might be able to proceed.

Execution of *signal_all(c)* awakens all the processes waiting on the condition variable *c*. In particular, its effect is the same as executing *signal(c)* 'n' number of times where 'n' is the number of processes in the delayed queue.

## 6.3 Implementation Using a Kernel

Monitors can be implemented directly by a kernel. The implementation of monitors using a kernel is more efficient than the implementation using semaphores. The following implementation is described in [4].

We assume that each process has a descriptor, and the descriptors of processes that are to execute are linked together on a ready queue. Also, to implement monitors, we add primitives for monitor entry, exit and each of the operations on the condition variables. Primitives are also needed to create descriptors for each monitor and each condition variable. These are nothing but defining and initializing the descriptors for each monitor and each condition variable.

Each monitor descriptor contains a lock and an entry queue of descriptors of processes waiting to enter the monitor. The lock is used to ensure mutual exclusion. When the lock is set, exactly one process is executing in the monitor; otherwise no process is executing in the monitor.

The descriptor of a condition variable contains the head of a queue of descriptors of processes waiting on that condition variable. Thus, every process descriptor-except those of executing processes- is linked to either the ready list, a monitor entry queue, or a condition variable delay queue.

The monitor entry primitive *enter(name)* finds the descriptor for monitor *name*,

then either sets the monitor lock and allows the executing process to proceed or blocks the process on the monitor entry queue. The monitor exit primitive *exit(name)* either moves one process from the entry queue to the ready list or clears the monitor lock.

The *wait(c)* statement is implemented by invoking the kernel primitive *wait(name,cname)* and the *signal(c)* statement is implemented by invoking the kernel primitive *signal(name,cname)*. In both primitives *name* is the name of the monitor descriptor and *cname* is the name of the condition variable descriptor. Execution of *wait()* delays the executing process on the specified condition variable queue and then either awakens some process on the monitor entry queue or clears the monitor lock. Execution of *signal()* checks the condition variable queue. If it is empty, the primitive simply returns; otherwise, the descriptor at the front of the condition variable queue is moved to the of the monitor entry queue.

Since a process that calls *wait()* exits the monitor, the wait primitive simply calls the exit primitive after blocking the executing process.

Figure 6.1: Monitor Kernel Primitives [4].

```
procedure enter (name: monitor_index)
        find descriptor for monitor name
        if lock = 1 → insert descriptor of executing at the end of entry queue
                executing := 0
        [] lock = 0 → lock := 1
        fi
        call dispatcher()
end
procedure exit (name: monitor_index)
        find descriptor for monitor name
        if entry queue not empty →
                move process descriptor form front of the entry queue to rear of ready list.
        [] entry queue empty → lock := 0          #clear the lock
        fi
        call dispatcher()
end
procedure wait (name: monitor_index; cname: condvar_index)
        find descriptor for the condition variable cname
        insert descriptor of executing at the end of the delay queue; executing := 0
        call exit(name)
end
procedure signal(name: monitor_index; cname: condvar_index)
        find descriptor for monitor name
        find descriptor for condition variable cname
        if delay queue not empty →
                move process descriptor from front of delay queue to rear of entry queue.
        fi
        call dispatcher()
end
```

# Chapter 7

# Implementation of Monitors

## 7.1 Data Structures

The following two structures are the only data structures added to the existing kernel. These structures are defined in the file */include/mon/mon.h*.

### 7.1.1 Monitor Descriptor

```
typedef struct {
        UINT8 lock;
        TID first;
        TID last;
} MON;
```

The *MON* is for the application user to declare and define monitor in their application at the user level. The first field *lock* determines whether the monitor is locked or not. That is, it determines whether there is a thread running in the monitor. The lock is set to *1* before a thread enters the monitor and it is set to *0* once the thread leaves the monitor. A thread $t$ can enter a monitor $m$ if and only if the *m.lock* is *0*(zero).

The other member variables *first* and *last* are used to maintain the list of threads waiting for their chance to enter the monitor. The data type TID is nothing

but an integer type. The variable *m.first* specifies the first thread waiting. It is NULL when there is no thread waiting to enter the monitor. The list is maintained using another variable *th_next[]* defined in the files *fp/kern.h* or *srpt/kern.h*. Using the *m.first* and *th_next[]* the second thread can be found. The second thread is *th_next[m.first]*. In the same way the list can be traversed. Finally, *th_next[m.last]* is NULL to specify the end of the list.

### 7.1.2 Condition Variable Descriptor

```
typedef struct {
        TID first;
        TID last;
} CONDVAR;
```

The other structure *CONDVAR* is used to define the condition variables in a monitor. In practice, each condition variable is part of a monitor. Since each monitor can have a variable number of condition variables, there are two ways of implementing condition variables. A monitor can have a linked list of condition variables or a new structure can be declared to define the condition variables. In the later case, user has to keep track of the condition variables belonging to each monitor. In the implementation described, a new structure is declared to define the condition variables.

Each condition variable has two member variables *first* and *last*, which are used to maintain the list of threads waiting on the condition variable. Both are set to NULL, if there is no thread waiting on the condition variable. Otherwise *first* is the first thread waiting and *last* is the last thread waiting.

## 7.2 Kernel Primitives

Below is a list of the additional kernel primitives used in the implementation of monitor primitives.

***void hal_begin_primitive(void)***

This function is called as the first instruction in all the primitives to prepare the environment for the kernel. The function disables all the interrupts and the kernel primitive is executed as an indivisible unit.

***void hal_end_primitive(void)***

This function is called as the last instruction of every primitive, to restore the environment of the calling thread. Note that it is implementation defined if this function returns to the kernel primitive or to the calling thread. The function enables all the interrupts.

***TID rq_queryfirst (void)***

This function is used to get the first ready task in the ready queue without extracting it.

***TID stk_queryfirst (void)***

This function is used to get the first stacked task (running task) queue without extracting it.

***void stk_getfirst (void)***

This function is used to extract the running task from the stack. The second task on the stack becomes the first one.

***void rq_insert (TID t)***

The function inserts a task/thread $t$ into the ready queue. The thread $t$ is inserted at its position in the ready queue depending on its priority. The function varies depending on the kernel used. The ERIKA supports two kernels namely FP Kernel and SRPT Kernel. The scheduling algorithm used by the FP Kernel is Fixed Priority with Preemption Threshold Scheduling Algorithm. The scheduling algorithm used by the SRPT Kernel is the Stack Resource Policy with Preemption Threshold Scheduling Algorithm.

***void hal_stkchange (TID t)***

This function is called when we need to switch from a frame to another frame that is already saved. The function saves the context of the current thread and prepares the system to switch to the context of the new thread $t$ passed as an argument.

***TID rq2stk_exchange (void)***

This function extracts the first task from the ready queue and inserts the extracted task on the top of the stack. The function returns TID of the extracted task.

***void hal_ready2stacked (TID t)***

This function is called when a context of a thread have to be saved. The context of the thread $t$ is saved on its user stack and the thread $t$ starts execution from the starting instruction once the function is returned.

***void sys_scheduler(void)***

This primitive simply executes a preemption check to see if the highest priority thread in the ready queue has higher priority than the running thread. In that case, the highest priority thread in the ready queue preempts the running thread. Otherwise nothing happens.

# Chapter 8

# Monitor Interface

The primitives described in this section cover the monitor mechanism interface that can be used by ERIKA applications. These primitives can be used both for process synchronization and mutual exclusion. There is no limit on the number of monitors the application can create.

*Note:* Monitor primitives can only be used with a multi-stack HAL, because these primitives are blocking.

To use a monitor, the user must include the header file *mon.h* in their application which is in the directory */include/mon/mon.h*.

## 8.1 Monitor/Condition Variable Initialization

A monitor can be defined and initialized as shown in the example below:

*MON mon1 = {0, NIL, NIL};*

In this example, *mon1* is initialized to zero, which means no task is running in the monitor and is free. The other two fields initialize the list of the tasks waiting to enter the monitor to be NULL.

A condition variable can be defined and initialized as shown below:

*CONDVAR condvar = {NIL, NIL};*

The list of tasks waiting on the condition variable condvar is initialized to NULL.

## 8.2 Another Example

This example describes a thread that uses a monitor to access a critical section.

```
MON mon = {0, NIL, NIL};
CONDVAR oktoread = {NIL, NIL};
void thread_example (void){
        ...
        ...
        /* The task enters a critical section protected by a mon monitor */
        mon_enter(&mon);
        while (nw > 0)
              mon_wait (&mon, &oktoread);
        nr = nr + 1;
        mon_exit (&mon);
        /* The task leaves the critical section allowing other tasks to enter
           the monitor */
        ...
        ...
}
```

## 8.3 The Functions

***void mon_enter (MON *mon)***

This function is used to lock the monitor by setting the lock variable associated with the monitor *mon*. If the lock value is already one, then the calling task does not return from the call to *mon_wait()* until it has been set to zero by another thread calling *mon_exit()*. The task has to call this function every time it enters

the monitor. This function makes the monitor busy and won't allow other tasks to enter the monitor.

***void mon_exit (MON *mon)***

This function is used by the tasks to exit the monitor. It checks to see whether there is any other task waiting to enter the monitor. If there is a task waiting to enter the monitor then it wakes up the task and compares the priorities of the woke-up task and the calling task to find out which one is to be scheduled. The function schedules the highest priority task. If there is no task on the monitor list, then it clears the lock associated with the monitor enabling others to know that the monitor is free and call the dispatcher to schedule the highest priority task.

***void mon_wait (MON *mon, CONDVAR *cond)***

This function is called by the tasks to wait on a condition variable cond associated with the monitor *mon*. The calling task is blocked and is added to the end of the list of tasks waiting on the condition variable. Since the calling function is blocked, it must leave the monitor, *mon_exit(mon)* is called as the last instruction in the function.

***void mon_signal (MON *mon, CONDVAR *cond)***

This function is used to signal a task waiting on a condition variable cond. It checks to see if the list associated with the condition variable is empty or not. If its empty it does nothing except scheduling a highest priority task. If it is not empty then it moves the task from the front of the delay queue associated with the condition variable (*cond*) to the rear of the entry queue associated with the monitor (*mon*) and schedules the highest priority task.

***void mon_signalall(MON *mon, CONDVAR *cond)***

This function is similar to the *mon_signal()* except that it moves all the tasks form the delay queue to the rear of the entry queue. It makes the delay queue associated with the condition variable empty. Finally, it checks the priorities of the tasks and schedules the highest priority task.

# Chapter 9

# Conclusion

Monitors are program modules that provide a strcutured approach for process synchronization. Assuming all shared variables are within monitors, two processes can interact only by calling procedures in the same monitor. Monitors makes a concurrent program easier to develop and understand [4].

The distribution of ERIKA doesn't provide monitors for the applications. It only provides mutual exclusion primitives and semaphore primitives. This document provides a structured approach for implementing monitors for ERIKA, which enables the users to declare, define and use monitors in their applications. Using this monitor module, real-time concurrent aaplications can be developed easily.

# Bibliography


[1] Paolo Gai and Alessandro Colantonio. **ERIKA User Manual (draft version)**, RETIS Lab, Pisa-ITALY, 2002.

[2] Paolo Gai. **ERIKA Mono-Stack Documetation Notes**, RETIS Lab, Pisa-ITALY, 2000.

[3] Paolo Gai. **ERIKA Multi-Stack Documetation Notes**, RETIS Lab, Pisa-ITALY, 2000.

[4] Gregory R. Andrews. **Concurrent Programming - Principles and Practice**, Addison-Wesley Publishing Company, 1991.

[5] Hoare C.A.R. **Monitors: an operating system structuring concept**, Comm. ACM 17,10(October), 1974.

[6] Brian W. Kernighan and Dennis M. Ritchie. **The C Programming Language**, Pentice Hall, 1988.

[7] SIEMENS AG. **Instruction Set Manual for the C166 Family**, Version 1.2, 12.97, 1997.

# Appendix A

# Source Code

The following sections show the files that were added in implementing the monitors for ERIKA. *mon.h* is put in the folder */include/mon*. The other files are put in the folder */src/mon*.

## A.1 mon.h

```
/*
 * Monitors for ERIKA (Embedded Real tIme Kernel Architecture)
 * Author: Sathish Kumar R. Yenna <sathish{AT}ksu.edu>
 * Dept. of Computing and Information Systems
 * Kansas State University
 *
 */

#ifndef __INCLUDE_MON_MON_H__
#define __INCLUDE_MON_MON_H__

/* Monitors
   ----

   This file declares the E.R.I.K.A. monitors and conditional variables.
   A monitor is contained in a data structure called MON and a condition
   variable is contained in a data structure called CONDVAR.
   That structure can be initialized statically (recommended), or
   dynamically using the macro mon_init.

   These functions can ONLY be used with a multistack HAL or similar,
   because these monitor primitives are BLOCKING primitives.

*/
```

```
// The monitor descriptor
typedef struct {
  UINT8 lock;
  TID   first;
  TID   last;
} MON;

// The conditional variable
typedef struct {
  TID first;
  TID last;
} CONDVAR;


#ifndef __PRIVATE_MON_ENTER__
void mon_enter(MON *m);
#endif

#ifndef __PRIVATE_MON_EXIT__
void mon_exit(MON *m);
#endif

#ifndef __PRIVATE_MON_WAIT__
void mon_wait(MON *m, CONDVAR *c);
#endif

#ifndef __PRIVATE_MON_SIGNAL__
void mon_signal(MON *m, CONDVAR *c);
#endif

#ifndef __PRIVATE_MON_SIGNALALL__
void mon_signalall(MON *m, CONDVAR *c);
#endif

#endif
```

## A.2 enter.c

```
/*
 * Monitors for ERIKA (Embedded Real tIme Kernel Architecture)
 * Author: Sathish Kumar R. Yenna <sathish{AT}ksu.edu>
 * Dept. of Computing and Information Systems
 * Kansas State University
 *
 */

#include <config.h>
#include <kernel.h>
#include <mon/mon.h>
```

```
void mon_enter(MON *m)
{
  TID current;

  hal_begin_primitive();

  if (m->lock) {
        /* The running task blocks:                                    80
                - It must be removed from the stacked queue
                - and then it must be inserted at the end of the monitor
                  entry queue */

        /* get the running task */
    current = stk_queryfirst();

        /* extract the task from the stk data structure */
    stk_getfirst();
                                                                        90
    /* The task state switch from STACKED TO BLOCKED */
    th_status[current] = BLOCKED;

    /* reset the thread priority bit in the system_ceiling */
    sys_ceiling &= ~th_dispatch_prio[current];

    if (m->first != NIL)
      // the monitor queue is not empty
      th_next[m->last] = current;
    else                                                               100
      // the monitor queue is empty
      m->first = current;

    m->last = current;
    th_next[current] = NIL;

#if defined(__FP__)
    /* check if there is to schedule a ready thread or pop a preempted
     * thread
     */                                                               110
    if (rq_queryfirst() == NIL ||
        sys_ceiling >= th_ready_prio[rq_queryfirst()])
#else
    // check if there is to schedule a ready thread or pop a preempted
    // th_absdline[stk_queryfirst()] <= th_absdline[rq_queryfirst()]
    // see also src/srpt/thendin.c
    if (rq_queryfirst() == NIL ||  //this test work also for the dummy task!
        (stk_queryfirst() != NIL &&
          (
            (signed)(th_absdline[stk_queryfirst()] -                   120
                th_absdline[rq_queryfirst()]) <= 0
```

```
            || sys_ceiling >= th_ready_prio[rq_queryfirst()]
          )
        )
      )
#endif
    {
        /* we have to schedule an interrupted thread that is on the top
         * of its stack; the state is already STACKED! */
        hal_stkchange(stk_queryfirst());                                    130
    }
    else {
        /* we have to schedule a ready thread that is not yet on the stack */
        th_status[rq_queryfirst()] = STACKED;
        sys_ceiling |= th_dispatch_prio[rq_queryfirst()];
        hal_ready2stacked(rq2stk_exchange());
    }
  }
  else
        m->lock = 1;                                                        140

  hal_end_primitive();
}
```

## A.3 exit.c

```
/*
 * Monitors for ERIKA (Embedded Real tIme Kernel Architecture)
 * Author: Sathish Kumar R. Yenna <sathish{AT}ksu.edu>
 * Dept. of Computing and Information Systems
 * Kansas State University
 *
 */                                                                         150

#include <config.h>
#include <kernel.h>
#include <mon/mon.h>

void mon_exit(MON *m)
{
  TID newthread;

  hal_begin_primitive();                                                    160

  if (m->first != NIL) {
    // wake up blocked thread
    newthread = m->first;
    if ((m->first = th_next[newthread]) == NIL)
      m->last = NIL;
```

```
        /* check for preemption */
#if defined(__FP__)
    if (sys_ceiling < th_ready_prio[newthread]) {
#else
    if (stk_queryfirst() == NIL ||      /* dummy task! */
       ((signed)(th_absdline[stk_queryfirst()] -
                 th_absdline[newthread]) > 0
         && sys_ceiling < th_ready_prio[newthread])) {
#endif
      /* we have to schedule the blocked thread */
      th_status[newthread] = STACKED;
      sys_ceiling |= th_dispatch_prio[newthread];

      // insert the extracted task on the top of the stack
      th_next[newthread] = stk_queryfirst();
      stkfirst = newthread;

      hal_stkchange(newthread);
    }
    else {
      th_status[newthread] = READY;
      rq_insert(newthread);
    }

    //Insert in the Ready Queue
    //th_status[newthread] = READY;
    //rq_insert(newthread);
  }
  else{
    m->lock = 0;
    /* Call the dispatcher */
    sys_scheduler();
  }

  hal_end_primitive();
}
```

## A.4 wait.c

```
/*
 * Monitors for ERIKA (Embedded Real tIme Kernel Architecture)
 * Author: Sathish Kumar R. Yenna <sathish{AT}ksu.edu>
 * Dept. of Computing and Information Systems
 * Kansas State University
 *
 */

#include <config.h>
#include <kernel.h>
```

```
#include <mon/mon.h>

void mon_wait(MON *m, CONDVAR *c)
{
  TID current;

  hal_begin_primitive();                                          220

    /* The running task blocks:
          - It must be removed from the stacked queue
          - and then it must be inserted at the end of the condition
            variable delay queue */

    /* get the running task */
    current = stk_queryfirst();

    /* extract the task from the stk data structure */            230
    stk_getfirst();

    /* The task state switch from STACKED TO BLOCKED */
    th_status[current] = BLOCKED;

    /* reset the thread priority bit in the system_ceiling */
    sys_ceiling &= ~th_dispatch_prio[current];

    if (c->first != NIL)
      // the conditional variable delay queue is not empty          240
      th_next[c->last] = current;
    else
      // the conditional variable delay queue is empty
      c->first = current;

    c->last = current;
    th_next[current] = NIL;

    // Call the monitor exit primitive
    mon_exit(m);                                                  250

    /* No need of calling the hal_end_primitive()
       as it is called in mon_exit()  */
    // hal_end_primitive();
}
```

## A.5 signal.c

```
/*
 * Monitors for ERIKA (Embedded Real tIme Kernel Architecture)
 * Author: Sathish Kumar R. Yenna <sathish{AT}ksu.edu>
 * Dept. of Computing and Information Systems
```

```
 * Kansas State University                                          260
 *
 */

#include <config.h>
#include <kernel.h>
#include <mon/mon.h>

void mon_signal(MON *m, CONDVAR *c)
{
  TID tempthread;                                                   270

  hal_begin_primitive();

  if (c->first != NIL) {
    // the conditional variable delay queue is not empty

    //Move the task from front of delay queue to rear of entry queue
    tempthread = c->first;
        c->first = th_next[c->first];
                                                                    280
    if (m->first != NIL)
      // the monitor queue is not empty
      th_next[m->last] = tempthread;
    else
      // the monitor queue is empty
      m->first = tempthread;

    m->last = tempthread;
    th_next[tempthread] = NIL;
                                                                    290
    /* The task state switch from BLOCKED TO STACKED  */
    th_status[tempthread] = STACKED;

  }

#if defined(__FP__)
    /* check if there is to schedule a ready thread or pop a preempted
     * thread
     */
    if (rq_queryfirst() == NIL ||                                   300
        sys_ceiling >= th_ready_prio[rq_queryfirst()])
#else
    // check if there is to schedule a ready thread or pop a preempted one
    //th_absdline[stk_queryfirst()] <= th_absdline[rq_queryfirst()]
    // see also src/srpt/thendin.c
    if (rq_queryfirst() == NIL ||  //this test work also for the dummy task!
        (stk_queryfirst() != NIL &&
          (
            (signed)(th_absdline[stk_queryfirst()] -
```

```
                th_absdline[rq_queryfirst()]) <= 0
            || sys_ceiling >= th_ready_prio[rq_queryfirst()]
          )
        )
      )
#endif
    {
        /* we have to schedule an interrupted thread that is on the top
         * of its stack; the state is already STACKED! */
        hal_stkchange(stk_queryfirst());
    }
    else {
        /* we have to schedule a ready thread that is not yet on the stack */
        th_status[rq_queryfirst()] = STACKED;
        sys_ceiling |= th_dispatch_prio[rq_queryfirst()];
        hal_ready2stacked(rq2stk_exchange());
    }

  hal_end_primitive();
}
```

## A.6 signalall.c

```
/*
 * Monitors for ERIKA (Embedded Real tIme Kernel Architecture)
 * Author: Sathish Kumar R. Yenna <sathish{AT}ksu.edu>
 * Dept. of Computing and Information Systems
 * Kansas State University
 *
 */

#include <config.h>
#include <kernel.h>
#include <mon/mon.h>

void mon_signalall(MON *m, CONDVAR *c)
{
  TID tempthread;

  hal_begin_primitive();

  while (c->first != NIL) {
    // the conditional variable delay queue is not empty

    //Move the task from front of delay queue to rear of entry queue
    tempthread = c->first;
        c->first = th_next[c->first];

    if (m->first != NIL)
```

```
      // the monitor queue is not empty
      th_next[m->last] = tempthread;
    else
      // the monitor queue is empty
      m->first = tempthread;                                          360

    m->last = tempthread;
    th_next[tempthread] = NIL;

    /* The task state switch from BLOCKED TO STACKED  */
    th_status[tempthread] = STACKED;

  }

#if defined(__FP__)                                                   370
    /* check if there is to schedule a ready thread or pop a preempted
     * thread
     */
    if (rq_queryfirst() == NIL ||
        sys_ceiling >= th_ready_prio[rq_queryfirst()])
#else
    //check if there is to schedule a ready thread or pop a preempted one
    //th_absdline[stk_queryfirst()] <= th_absdline[rq_queryfirst()]
    // see also src/srpt/thendin.c
    if (rq_queryfirst() == NIL ||  //this test work also for the dummy task! 380
        (stk_queryfirst() != NIL &&
          (
            (signed)(th_absdline[stk_queryfirst()] -
                th_absdline[rq_queryfirst()]) <= 0
           || sys_ceiling >= th_ready_prio[rq_queryfirst()]
          )
        )
       )
#endif
    {                                                                 390
        /* we have to schedule an interrupted thread that is on the top
         * of its stack; the state is already STACKED! */
        hal_stkchange(stk_queryfirst());
    }
    else {
        /* we have to schedule a ready thread that is not yet on the stack */
        th_status[rq_queryfirst()] = STACKED;
        sys_ceiling |= th_dispatch_prio[rq_queryfirst()];
        hal_ready2stacked(rq2stk_exchange());
    }                                                                 400

  hal_end_primitive();

}
```